Aalto University
School of Science
Master's Programme in Security and Cloud Computing

Ngadhnjim Plaku

Online Platform for Interactive Tutorials: Provisioning Virtual Environments

Master's Thesis
Espoo, July 31, 2020

DRAFT! — July 31, 2020 — DRAFT!

Supervisors: Professor Mario Di Francesco, Aalto University
Professor Pietro Michiardi, EURECOM
Advisor: Professor Mario Di Francesco

Aalto University
School of Science
Master's Programme in Security and Cloud Computing

ABSTRACT OF
MASTER'S THESIS

Author: Ngadhnjim Plaku

Title:
Online Platform for Interactive Tutorials: Provisioning Virtual Environments

Date:	July 31, 2020	**Pages:**	50
Major:	Security and Cloud Computing	**Code:**	SCI3084
Supervisors:	Professor Mario Di Francesco Professor Pietro Michiardi		
Advisor:	Professor Mario Di Francesco		

Traditionally, whenever students learn a new technology, they need to either set up their working environments on their own machines or go to physical laboratories provided by the teaching institutions. In the first case, the setup of the needed tools is cumbersome or even impossible on all personal computers; in the second case, laboratories for certain topics are not feasible due to the threat to the stability and security of the system. Recently, virtualization has been extensively used to provide dedicated environments with full control over the system and to create interactive tutorials that engage the student with the learning resources through a web browser. However, most of the tools that are publicly available are built for specific purposes and are not extensible. There are many situations where organizations need customized solutions that give them full control over the system to offer better support as well as progress tracking and automated assessment. This thesis describes the implementation of OnPIT, an online platform for interactive tutorials. Different from the existing work, OnPIT is based on software containers. Moreover, it provides access to learning environments through a web browser, side by side with the tasks to be completed during a tutorial. Furthermore, OnPIT allows the creation of new tutorials that are preconfigured with the necessary tools, it can be scaled based on the demand and the available resources, and it also supports automated assessment. The thesis overviews the overall design and implementation of OnPIT. Afterwards, it details the provisioning of dedicated environments based on software containers, including the security issues and technology limitations as well as possible solutions to overcome them.

Keywords: virtualization, containerization, provisioning, online learning, security

Language: English

Acknowledgements

I would like to thank Professor Mario Di Francesco for his support and feedback throughout the work of this thesis. I am also grateful to my colleagues for their friendship and support during the implementation of this project.

Finally, I would like to thank my family and friends for their continuous moral support and motivation.

Thank you!

Espoo, July 31, 2020

Ngadhnjim Plaku

Abbreviations and Acronyms

API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
CNI	Container Network Interface
CPU	Central Processing Unit
CRI	Container Runtime Interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
I/O	Input/Output
IP	Internet Protocol
KVM	Kernel-based Virtual Machine
OCI	Open Container Initiative
OnPIT	Online Platform for Interactive Tutorials
OS	Operating System
REST	Representational State Transfer
RST	reStructuredText
SQL	Structured Query Language
SSH	Secure Shell
UID	Unique Identifier
URL	Uniform Resource Locator
VM	Virtual Machine
VMM	Virtual Machine Monitor
WS	WebSocket

Contents

Chapter 1

Introduction

Technological advancements have pushed learning towards digitalization. Recently, online learning systems have been developed for a set of use cases ranging from professional trainings in industry to distance learning in academia [40]. The benefits of online learning systems include lower cost, easier scaling and infrastructure upgrade as well as flexibility with remote and self-paced learning [19].

Learning computer science in higher education is typically supported by hands-on exercises. These exercises are performed either on student's personal computers or on shared computers offered by the academic institution. There have been many efforts to shift from such a traditional approach towards new online tools that provide the same capabilities as well as personalized interaction and learning paths [40]. This has started with the advent of web technologies that enabled online tutoring based on demand. These solutions offer interactive tutorials mainly targeting programming languages, by allowing the user to write the code in the browser, compiling it on the back end and presenting the output back to the user [26].

The advent of virtualization technologies enabled both better resource utilization and the creation of isolated instances of an operating system. This led to new opportunities for learning systems to provide a dedicated machine to a student for learning purposes [27, 30]. These machines are typically accessed through some form of remote connection, such as SSH. However, an increasing number of applications are integrating an interactive terminal into a web page. This allows the combination of the terminal and the teaching material to create a better learning experience for the students.

These dedicated environments are built for specific courses and provide the infrastructure and the tools needed. This is used for a wide range of topics in computer science from operating systems and system administration to networking and security [28]. However, computer science is a fast-evolving

field and new topics such as cloud computing and machine learning are also finding their way into online learning systems with platforms such as Katacoda[1].

1.1 Motivation

Online learning systems are built by institutions for their own needs or for dedicated purposes. Most of the systems focus on offering solutions for a specific course, technology or platform and the main targets are programming languages and web technologies. Even the systems that provide access to dedicated infrastructure through virtualization do not provide a full set of features that would make them fully reusable and extensible. Creating new content and customizing course environments is not easy when offering a solution based on virtual machines. Furthermore, relying on third party services is not feasible for many institutions that need custom features and in-house support, limited access to enrolled students and progress tracking.

1.2 Contribution

This thesis presents the design and implementation of OnPIT, an online learning platform for interactive tutorials. The platform offers tutorials accessible through a web browser, accompanied with isolated and dedicated environments to each student. It gives access to these environments through a terminal in the web browser, as it were a virtual machine inside the browser. In addition, OnPIT is scalable, allows easy creation of content and customization of the environments, restricting access only to enrolled students as well as customized progress tracking.

This thesis establishes the following contributions:

- realizes the feasibility of provisioning isolated environments through containerization;
- discusses the limitations of container technology in terms of extensibility;
- presents the trade-offs between security and extensibility when using software containers;
- introduces alternative solutions that combine containers with VMs and evaluates their interoperability with the current implementation;

[1] `https://www.katacoda.com/`

- compares the considered solutions in terms of performance, extensibility and security.

1.3 Structure of the Thesis

The rest of this thesis is organized as follows. Chapter 2 overviews the technologies and major concepts relevant for this thesis. Chapter 3 describes the design and implementation of OnPIT, an online platform for interactive tutorials. Chapter 4 explains how this platform provisions the learning environments for the students. Chapter 5 evaluates the provisioning tools used in terms of security and interoperability. Finally, Chapter 6 concludes the thesis and suggests potential future work.

Chapter 2

Background

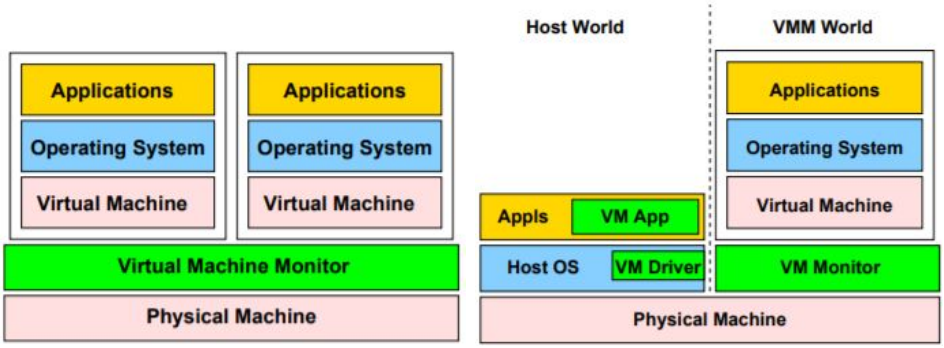
This chapter overviews the major technologies this thesis is built upon. It first introduces the traditional virtualization technology also known as hypervisor-based virtualization. It then presents the common components and concepts related to the container technology. Moreover, it discusses Docker and Kubernetes, the most popular software tools for creating and managing containers. Finally, the chapter considers the security features provided by these technologies.

2.1 Virtualization

Virtualization was introduced as a solution to overcome issues in allocating resources to multiple applications running on the same physical machine at the same time. One such example is a single application taking up most of the resources, thus degrading the performance of the other applications running on the same machine. Before the introduction of virtualization, it was not possible to limit the resources to a given application. The only solution to this problem was to run each application on its own physical machine. However, this solution came with additional maintenance costs and it was not scalable.

As an alternative, virtualization provides isolation to the applications while they still run on the same physical resources, allows better utilization as well as scalability, and has a lower cost [31].

Typically, virtualization refers to a software layer wrapping an operating system to make it appear as a standalone physical machine [33]. This layer provides to the operating system the same environment as if it were a real device. These virtual environments are created by a piece of software called a Hypervisor or Virtual Machine Monitor (VMM) [34]. These environments



(a) Type I (b) Type II

Figure 2.1: Types of Virtual Machine Monitors [37]

are known as *virtual machines*. Each of the created virtual machines can run their own operating system: even though they share the physical resources of the host machine, they are logically as separate as if they were different physical machines. As a result, this technology can be used for providing independent and isolated environments to different applications as well as to reduce the number of physical machines needed.

There are two main types of hypervisors: Type I (Bare Metal) and Type II (Hosted), [36], illustrated in Figure 2.1.

- A type I hypervisor runs as an operating system or kernel directly on top of the bare machine. The hypervisor has the mechanisms to schedule and allocate the system resources to the virtual machines. Products built according to this approach include Xen [21], VMware ESX [32] and Microsoft Hyper V [38].
- A type II hypervisor runs as an application on top of a host operating system. It provides services to support virtualization, but it does not schedule and allocate the system resources. Instead, this is handled by the host operating system. Products built according to this approach include QEMU [22], VMWare Workstation [37] and Microsoft Virtual PC [29].

2.2 Containerization

A *software container* is a lightweight alternative to virtualization [39]. Like virtual machines, containers offer resource isolation and allocation, however,

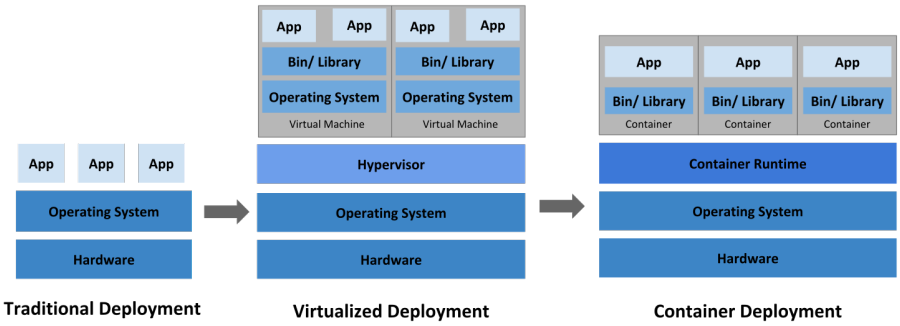


Figure 2.2: Container evolution [17]

they do not virtualize the hardware but are software units sharing the same operating system [23]. Such a unit represents a package of the application code and all its dependencies. Packaging applications as containers makes them more lightweight and efficient compared to virtual machines, since they do not contain an operating system image within. The decoupling of the software from its environment through containerization results in a consistent behavior of the applications no matter the infrastructure. Figure 2.2 illustrates the evolution from traditional deployment to virtualization and then to containers.

There are many advantages that have made containers popular: First, container images are easier to create and deploy using tools such as Docker. Since they do not need an operating system within, containers are lightweight and allow fast, reliable and frequent image build and deployment [39]. In addition, containers are more efficient with resource utilization, thus providing higher density of the applications. Since containers package both the application code and its dependencies, the applications running within containers are decoupled from infrastructure. Furthermore, Containers can run on many operating systems, on-premises, in the cloud or bare metal. Since the containers are software packages created using Linux features such as namespaces and cgroups, they can be monitored from the host system to ensure that they are healthy and to retrieve metrics about their resource utilization.

2.2.1 Container Images

A *container image* is a file that contains the executable code needed for creating a container at runtime. Container images are typically stored in a so-called registry server. They are pulled from such a registry and used locally to create and run containers. Almost all major tools and container

engines support a standardized format from the Open Container Initiative (OCI). This standard[1] defines the image format as a group of tar files, known as layers, and a `manifest.json` file which contains metadata.

2.2.2 Container Engines

A *container engine* is a software that handles requests from the command line interface as well as from container orchestration tools. When the given command requires running a container, the engine pulls the specified image from the registry server and runs the container through the container runtime. In addition, the engine decompresses the container image on the disk and prepares the settings and metadata that the container runtime needs, based on the default settings from the container image and the user input. The major container engines include Docker[2], LXD[3] and RKT[4]. Using the OCI standard for container images enables interoperability between the container engines [1].

2.2.3 Container Runtimes

A *container runtime* is a software that executes containers and manages their lifecycle. A container runtime is a lower level component and it is typically used in a container engine and not standalone. Similar to container images, OCI specifies a standard for container runtimes. Moreover, they provide *runc*[5] as a reference implementation. This is the most widely used container runtime and many container engines, including Docker and CRI-O[6], rely on it. Other OCI-compliant runtimes include crun, railcar and Kata Containers [1]. The implementation of the OCI standard allows containers to run consistently across different container engines.

The container runtime is responsible for retrieving and acting based on the instructions of the container engine. Furthermore, it starts the containerized processes by communicating with the kernel through system calls.

[1] `https://github.com/opencontainers/image-spec`
[2] `https://docs.docker.com/engine/`
[3] `https://linuxcontainers.org/`
[4] `https://coreos.com/rkt/`
[5] `https://github.com/opencontainers/runc`
[6] `https://cri-o.io/`

2.2.4 Container Orchestration

Container *orchestration* is the process of automatic management of containers [1]. Orchestrations tools automate many tasks, such as provisioning and deployment of containers, scaling up or down, availability, resource allocation, health monitoring, and so on. Typically, container orchestration tools take a configuration file for each application that runs as a container. These files describe the desired state of the application container, including the image to be used, network settings and volumes to be mounted. When a new container needs to be deployed, the orchestrator schedules the container applications based on the predefined configuration settings.

There are many orchestration tools developed, such as Swarm, Mesos, and Kubernetes. However, Kubernetes is the most widely supported tool and is considered as the de facto standard.

2.3 Docker

Docker is an open source platform for developing, shipping and running containerized applications [3]. Such a platform provides the tools needed to manage the lifecycle of the containers. It allows to containerize and run an application as a unit in an isolated environment. Isolation, security, and less resource utilization as well as being lightweight compared to a hypervisor, enable users to run more containers than virtual machines. Furthermore, a Docker container is independent from the underlying infrastructure and this enables the users to run the application the same way no matter the deployment environment.

Docker uses a client-server architecture that consists of a server, a REST API and a command line interface (CLI) client. The CLI uses the Docker REST API to control or interact with the server. The server, also known as the Docker daemon (dockerd), creates and manages Docker objects, such as images, containers, networks, and volumes.

Figure 2.3 illustrates the architecture of Docker. Here the Docker client receives a command from the user and forwards it to the Docker daemon through the REST API. The Docker daemon listens for API requests and manages Docker objects accordingly. When an image is required as a result of a user command, it is pulled from a registry which can be private or public (e.g. Docker Hub), then it is stored in a local cache and used for creating a new container instance.

By default, the daemon listens for API requests through a unix domain socket. At the beginning, Docker was a monolithic application. Over time,

some functionalities were decoupled from the main project. This way containerd, a high-level container runtime, took the responsibility of managing the container lifecycle by passing the commands to the low level runtime, runc. Since then, using Docker indirectly calls containerd, which in turn calls runc. However, containerd can be used as a standalone runtime and it has its own CLI command ctr. Runc is the component that handles the container lifecycle management and it is the reference runtime implementation of the OCI specification. Figure 2.4 illustrates the path from Docker to the containers.

The most common Docker objects managed by the daemon are images and containers. Images act as templates for creating containers. Users can create their own images or use those created by others and made publicly available in a registry. Building an image is accomplished through a Dockerfile or by saving the state of a running container. The Dockerfile contains the instructions needed to create and run the image. Each instruction creates a so-called layer in the image. When an image is changed and rebuilt, only the layers which have been updated are rebuilt. On the other hand, a container is a runnable instance of an image. Containers can be managed using the Docker API or CLI. By default, a container provides some level of isolation from the host machine and other containers, but this can be changed based on the per-image settings and by configuration options provided upon creating or starting it.

Docker is written in the Go language and its functionality is based on several features of the Linux kernel. Docker Engine combines different features

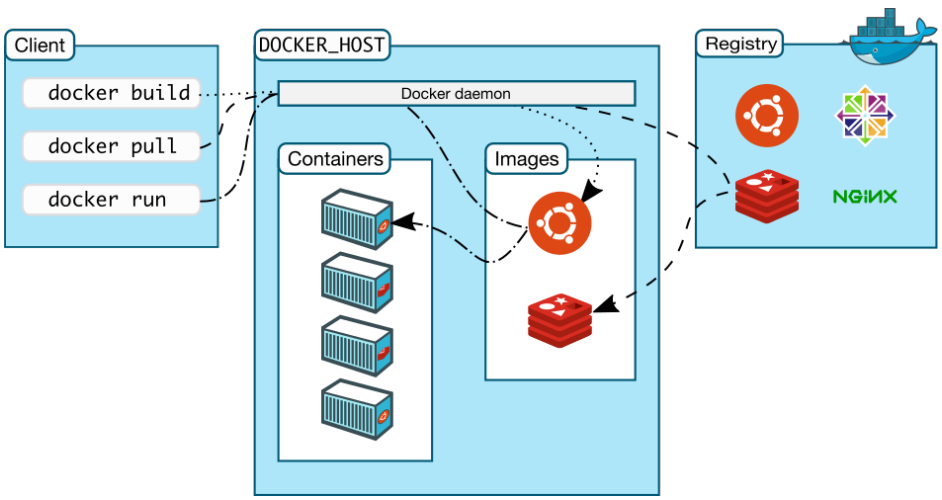


Figure 2.3: Docker architecture [3]

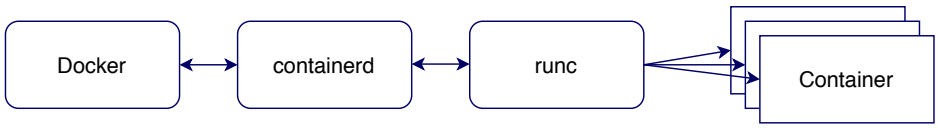


Figure 2.4: Docker components stack

of the Linux kernel – namely, namespaces, control groups, and UnionFS – into a wrapper called a container format. Each of them is described briefly below:

- *Namespace*[7] is a feature that makes the processes within it have an isolated view of a specific resource. Docker creates a set of namespaces for each container that is run. Such namespaces include: ProcessID, networking, InterProcess Communication, mount and Unix Timesharing System. Each of them helps containers to achieve their isolated view of the system. ProcessID namespace provides process isolation enabling processes in different namespaces to use the same PID number. Networking namespace isolates the networking interfaces. Similarly, InterProcess Communication namespace isolates IPC resources. Mount namespace manages the filesystem mount points for each namespace. Whereas Unix Timesharing System namespace isolates system identifiers between processes.

- *Control groups (cgroups)*[8] are used to limit the available resources to a specific container. This allows Docker to share hardware resources among many containers while having control over resource allocations for each of them.

- *Union file systems (UnionFS)*, are layer-based file systems, making them lightweight and fast. They are used by Docker for efficient storage of the container images. Docker creates a read-only layer for each instruction in the Dockerfile of an image. When a container is created, a new writable layer is added. All changes to a container are stored in this layer, thus enabling multiple containers to share the same underlying image [2].

[7] `https://man7.org/linux/man-pages/man7/namespaces.7.html`
[8] `https://man7.org/linux/man-pages/man7/cgroups.7.html`

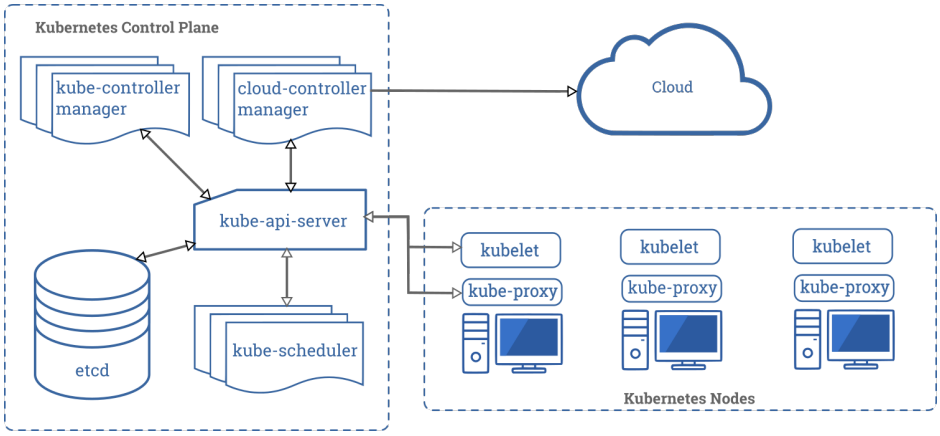


Figure 2.5: Components of Kubernetes [9]

2.4 Kubernetes

Kubernetes is an open-source container orchestration tool based on Google's internal system called Borg [25]. Since its introduction, Kuberentes has become the most widely used orchestrator. It can handle load balancing and expose a container to the outside world; it can also automate container deployment and removal, and adjust the allocated resources based on the desired state described by the user. Kubernetes relies on a cluster consisting of at least one worker machine, called a node, which can be a physical or virtual machine. Given a cluster of nodes and the instructions for the desired containers' state, Kubernetes automatically establishes and keeps that state with the optimal usage of resources. Even if a container fails or is not responsive, Kubernetes kills and replaces it automatically until it comes back to a normal state. Furthermore, Kubernetes offers storage and management for sensitive information, such as passwords, tokens, and keys.

2.4.1 Main Components

Usually a Kubernetes cluster consists of multiple nodes to provide high availability. Inside these worker nodes, Kubernetes runs the application workloads as *pods*. Pods are the smallest deployment units in Kubernetes, and they consist of one or more containers with shared resources [13].

The worker nodes and pods in the cluster are managed by a *control plane* which comprises many components. These components have specific roles when it comes to responding to cluster events and maintaining the desired

state. Even though these components can run on separate machines, usually all of them reside on the same machine and this machine does not act as a worker node (i.e., does not run containers). Similarly, each worker node consists of a set of components which ensure that the pods that are assigned to them are running. Figure 2.5 shows a Kubernetes cluster, whose components are detailed below [9].

The control plane components are as follows:

- kube-apiserver is an implementation of an API server for Kubernetes. The API server has direct access to the cluster, and it exposes the Kubernetes API to users, tools and other components in the cluster. It is implemented as a RESTful API and stores the API objects in the etcd persistent storage [15, 24]. Command line tools such as kubectl and kubadm (commonly used by developers to interact with the Kubernetes API) make API calls to this server on behalf of the user;

- etcd is a key value storage used for saving all Kubernetes objects. It provides reliable and consistent storage for distributed systems;

- kube-scheduler is a component that assigns a node to a newly created pod. It makes the node choice based on resource requirements, and other constraints and specifications;

- kube-controller-manager is a component that runs controllers. Each controller is assigned to a Kubernetes resource type and is responsible for sending messages to the API server to bring the current state closer to the desired state [9]. The Kubernetes controllers include node controller, which responds when a node goes down; replication controller, which maintains the correct number of pod replicas; endpoints controller, which populates the Endpoints object; and Service Account Token controllers, which create default accounts and access tokens;

- cloud-controller-manager is a component that links a cluster with a cloud provider API and separates the components interacting with that cloud from those that interact only with the cluster [9]. The controllers belonging to this component include: node controller, which determines whether a node has been deleted if it stops responding; route controller, which sets up routes in the cloud infrastructure; and service controller, which creates, updates and deletes cloud provider load balancers.

The components that run on each node in the cluster are:

- kubelet ensures that containers are running in a Pod. It takes a set of PodSpecs and ensures that the containers described there are running and healthy;
- kube-proxy is a network proxy that maintains network rules on nodes, enabling communication between the cluster components and with external components;
- container runtime is the software that runs containers, such as Docker, containerd or CRI-O.

2.5 Security

Docker containers take advantage of namespaces to achieve their isolation [5]. Each of the used namespaces isolates a certain class of resources. For example, process ID namespace prevents processes running within a container from seeing or interacting directly with the processes running in another container, or in the host. Similarly, network namespace provides each container with its own network stack.

Control groups are another feature used by containers to control resources but they do not offer isolation. Instead, they help the system protect against denial of service attacks [5].

Since containers share the operating system with the host, most of the system-level tasks are delegated to the host system. As a result, in general containers can run their application with less capabilities and privileges than the host system. Docker allows the configuration of these capabilities depending on the needs of each use case.

On the other hand, Kubernetes builds on top of these container features and provides to the user a set of capability controls. These controls can be used to limit the resources used on a cluster, define user privileges of each node, prevent loading of some kernel modules, set up network policies, authentication, role based access control, secrets management and more [12].

Chapter 3

Online Platform for Interactive Tutorials

Online Platform for Interactive Tutorials (OnPIT) is a system that enables students to carry out interactive tutorials through a web application [11]. This chapter describes the design and implementation of OnPIT. It first describes the system. Afterwards, it presents the implementation of the system architecture. The first component described is the web server and its features, which include a description of the course content management, the web terminal and its communication with the corresponding sandboxed environment, the automated assessment of the exercises as well as authentication and authorization of users. Finally, it discusses how data is stored in the system.

3.1 System Architecture

OnPIT has several components that enable it to provide interactive tutorials through a web browser, provision isolated environments for each user, automate the assessment of the tutorials and manage the content. Figure 3.1 overviews the system and its components.

The platform runs within a Kubernetes cluster. The cluster contains a web server and a database. It also launches containers which are used as environments for the interactive tutorials and allows their integration based on virtual machines. To retrieve the course content created by the teachers, it connects to a git server from which it receives updated content whenever it is changed. The platform is accessible only through the web server which accepts client requests over HTTP and WebSocket.

Currently, the system resides on Google Cloud Platform. However, it is

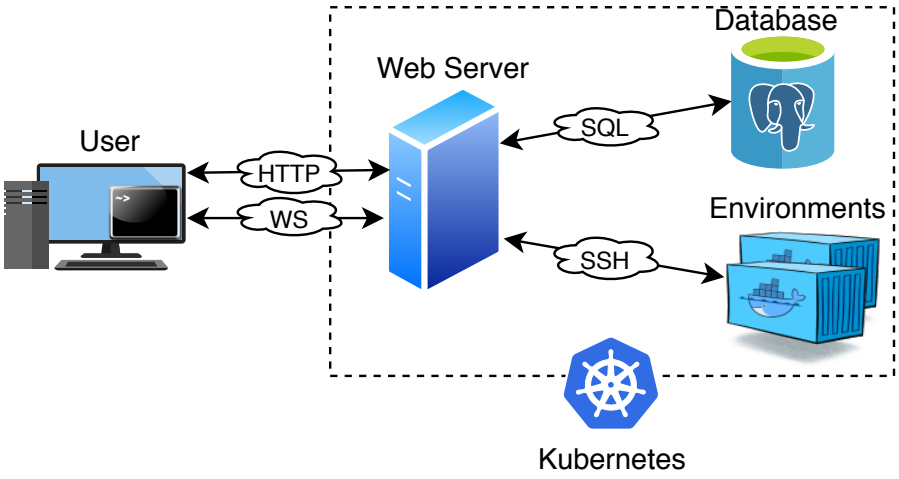


Figure 3.1: Architecture of the system

not dependent on a specific cloud provider and it can be deployed on any of them or even on-premise.

3.2 Web Server

The web server is the main component of the platform. It serves as the entry point for the user requests and communicates with all the other components of the system to handle tasks such as storing and retrieving the data, provisioning the learning environments and assessing the tutorials. The web server is implemented in Python[1] with Django[2] web framework. The main responsibility of the web server is to display the web application to the users. To do so, it needs to store the content and process it based on the functionality that the platform intends to provide.

The platform allows teachers to create their own courses. The content of the courses is created into files following a specific format (described in Section 3.2.1) and stored in a git repository. To add a course to the platform, the web server provides to the teachers a web interface where they give the information needed to retrieve the data from the repository.

The courses consist of one or more labs. Each lab is split to several steps that contain the teaching material or the instructions for the exercise. The teacher also provides some metadata about the courses and labs, and specifies

[1] `https://www.python.org/`
[2] `https://www.djangoproject.com/`

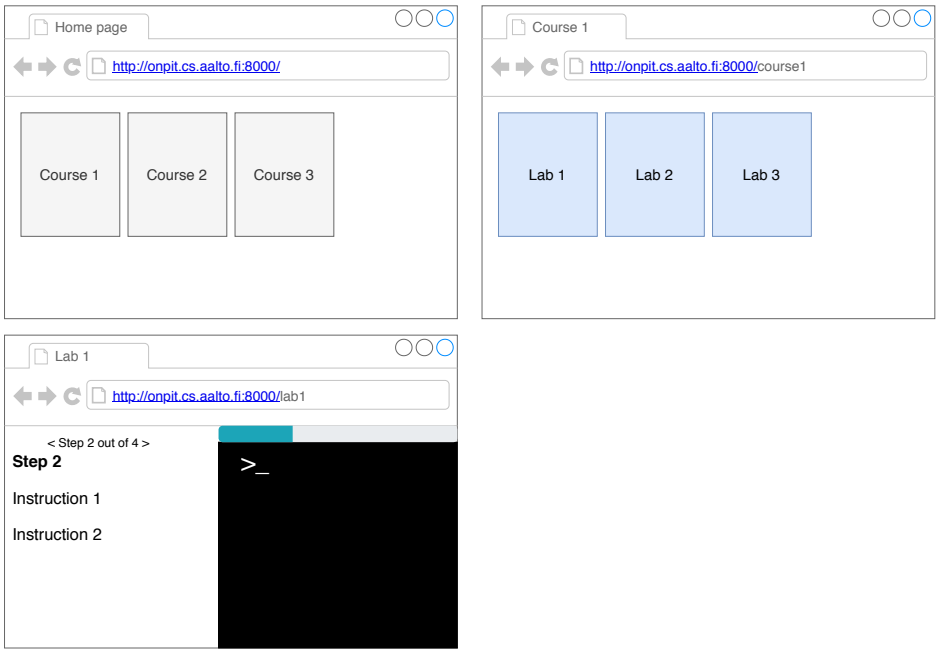


Figure 3.2: Workflow of a tutorial

how each step of a lab is automatically assessed. The web server takes all this information and transforms it into HTML. Figure 3.2 illustrates the workflow of a tutorial. The courses are listed in the main page and upon clicking any of them the user is redirected to a sub-page specific to that course, which lists the labs contained therein. When a lab is clicked, the server shows a terminal page which contains the instructions on one side and the terminal for completing the exercises on the other side. The web server also handles the binding of the terminal to a dedicated environment (described in Section 3.2.2) and instructs the provisioning of these environments (discussed in Chapter 4).

Whenever the student sends a command through the terminal, the server checks whether that command achieves the result expected from that step. The assessment rules for each step are specified by the teacher when creating the course, and this is further discussed on Section 3.2.3. If a step is completed successfully, this is shown by a progress bar on top of the terminal.

Finally, the web server also handles the authentication and authorization of users (discussed in Section 3.2.4).

3.2.1 Content Management

OnPIT enables teachers to create their own course content. Such a content is defined as a directory of RST[3] and YAML[4] files. A course has one or more labs, the content for each of them is in a sub-directory. Both course and lab directories contain a YAML file. An example YAML file for a lab is shown in Listing 1.

```
title: Title of the lab
description: description of the lab
details:
  intro:
    text: intro.rst
  steps:
  - name: step1
    title: Title for step 1
    text: step1.rst
  - name: step2
    title: Title for step 2
    text: step2.rst
  finish:
    text: finish.rst
grader: grader.py
initializationScript: setup.py
environment:
  image: ubuntu
  technology: container
```

Listing 1: Example YAML file for a lab

The YAML files contain metadata about the corresponding course or lab, such as the title and a description. In addition, the YAML file of the course lists the directory paths for each of its labs, while the YAML file of the lab lists each of the tutorial steps and their corresponding files. The textual content of each step is written using RST (reStructuredText) markup language, which is chosen because it has an easy syntax, it is extensible, and it can be converted to other formats. OnPIT automatically converts the content of these files into HTML and displays it during the tutorial steps without the need for further modifications.

[3] https://docutils.sourceforge.io/rst.html

[4] https://yaml.org/

A lab can also have two python files: a setup script, which is executed before the learning environment is displayed to the student, and a grading script, which contains the code for the automated assessment of each step (described on Section 3.2.3). In addition, a lab chooses one of the technologies and images supported by the system.

An example file hierarchy of a course directory is shown below:

```
course_directory
├── course_index.yaml
├── lab_name
│   ├── lab_index.yaml
│   ├── intro.rst
│   ├── step1.rst
│   ├── step2.rst
│   ├── finish.rst
│   ├── grader.py
│   └── setup.py
└── lab_name_2
    └── ...
```

OnPIT downloads courses from git repositories. To add a new course to OnPIT, a teacher has to `push` the content to a git repository and then give the repository URL through the web interface. In addition, OnPIT automatically gets a notification when the content is updated on the repository and performs a new `pull` to get the updates. This is achieved through webhooks[5]. When a course is created, OnPIT generates the necessary credentials and a URL that is given to the git host. When an event (e.g., `push`) is triggered, OnPIT receives an HTTP POST message to this URL and updates the local content to keep it on sync with the git repository. This enables the teachers to automatically update the course content without interaction with the platform itself.

3.2.2 Terminal Binding

The web terminal is implemented using a front-end component called *xtermjs*[6]. This terminal instance allows binding to the background process in the web server through the WebSocket protocol. The WebSocket protocol facilitates the data transfer between the browser and the server since it maintains a communication between the two and enables transfer in both directions.

[5] `https://docs.github.com/en/developers/webhooks-and-events/about-webhooks`

[6] `https://xtermjs.org/`

On the other end of the communication, the web server connects to the sandboxed environment through SSH communication. This way, the web server acts as an intermediary between the sandboxed environment (which is provided to a user) and the web browser. Any input from the terminal on the web browser is sent to the web server, which then forwards it to the environments over SSH. Similarly, any output stream coming from the environment, is forwarded to the front-end terminal which renders and displays it to the student.

3.2.3 Automated Progress Tracking

OnPIT allows teachers to automate the assessment of their courses. This is done by giving the instructions to the platform through a python script. This assessment script is located inside the directory of a lab. The assessment scripts contain the instructions for evaluating the successful completion of a step in the lab. These instructions are sent from the web server to the student environment through an SSH connection with root privileges. The instructions are executed with root privileges and can monitor the state changes on the system to track the achievement of the tutorial goals. In addition, the tutorial steps can be assessed based on the input stream which contains the command of the student. This allows the teacher to check that the student is using a specific command to achieve the result. The result from the execution of an assessment command is sent back to the web server, which stores the result and updates the web browser view for the student.

3.2.4 Authentication and Authorization

Authentication is the process of verifying the identity of a user, while authorization is the process of verifying that the user can perform an action. To use OnPIT, users have to be authenticated. The platform offers two methods of authentication: third party authentication with Google and built-in authentication, which is handled through Django on the web server.

Google authentication allows Aalto University members to authenticate to the platform by logging in to Google with their university accounts and is limited to users within the organization.

As an alternative, the users can register and login with the system's authentication. Django offers authentication features that handle user accounts, groups and permissions. OnPIT uses these features to provide user registration and login, as well as role-based access control.

OnPIT has two groups of users: teachers and students. Teachers can create and manage courses, as well as enroll their students to these courses.

On the other hand, students are allowed access only to the courses that they are enrolled into.

Necessary authorization checks are implemented to make sure that neither students nor teachers can access courses that they do not own or that they are not enrolled to. In addition, users are not allowed to carry actions that are not allowed by their roles (e.g., a student is not able to create courses).

3.3 Database

OnPIT needs a persistent storage for data such as user information and their progress on the assignments. In addition, data storage can also be used for the content of the courses, to facilitate the data retrieval and management.

The platform uses a relational database. This was chosen because OnPIT has simple and well-defined data structures and relations between them, as shown in Appendix A.

The chosen management system for the database is PostgreSQL[7], which runs inside a container within the Kubernetes cluster, and it communicates only with the web server, which inserts and consumes the data. PostgreSQL was chosen because it is major open source software and it is officially supported by Django. Thus, all the database operations take place through the Django integration, by working with classes and objects instead of SQL queries. Furthermore, Django maintains a persistent connection with the database to decrease response time without having to open a connection on each request.

[7] `https://www.postgresql.org/`

Chapter 4

Provisioning Student Environments

This chapter explains how OnPIT provisions the learning environments for students. The chapter starts with an overview of the possible implementation of such environments based on software containers. Afterwards, it presents some alternative solutions based on lightweight VMs and how they are integrated in the system.

4.1 Container-based Environments

The main components of OnPIT, including the web server and database, already exist inside a Kubernetes cluster. Therefore, building the learning environments on top of the same platform allows easier system integration and management of the environments from within the web server. This is achieved by running a container for each session of a lab and then passing the input/output streams between the container and the web terminal that is shown to the user, using the web server as an intermediary.

Implementing the environments based on this design has two main steps, as shown in Figure 4.1. First, the web server manages the lifecycle of the student containers based on the students' interaction with the system. Next, it obtains a terminal connection to the container and pass the input/output streams between the container and the student. To manage the lifecycle of the containers, the web server connects to the Kubernetes API Server. Upon receiving the instructions from the web server, Kubernetes creates or deletes a student's container. To establish a direct communication between the container and the web server, the container runs an SSH server to which the web server connects as a client.

This architecture allows the teachers to offer different container environments based on their course content. To do so, they need to specify the name

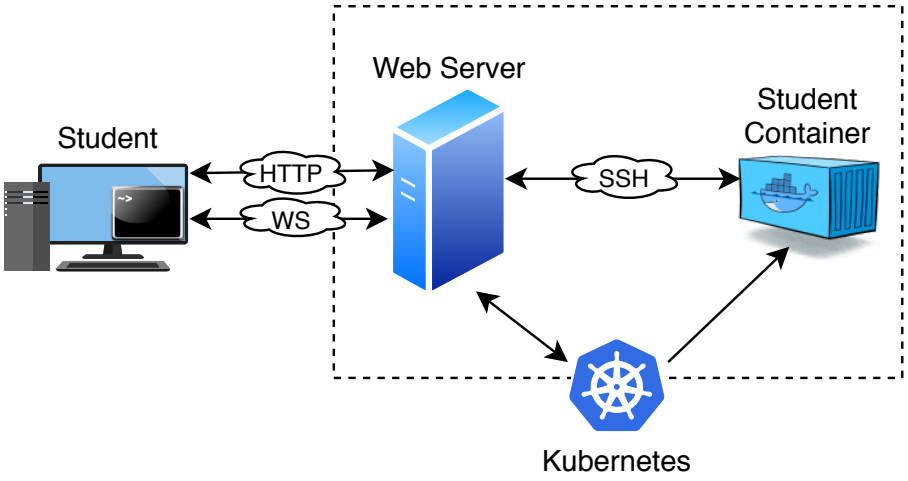


Figure 4.1: Container-based environments

of the container image for each lab; the image is then pulled from a container registry and launched when a student opens the lab. Docker Hub includes so-called *official images*[1] which provide the basic starting points for a number of projects, ranging from operating systems, to programming language runtimes and other services, and they are supported and maintained by the corresponding upstream projects. These images are a reliable option for base images. However, most of them need to be modified to fit the purposes of OnPIT (i.e., to allow access from the web server to the container).

The following sections analyze the potential use of Docker containers for more complex environments, namely Docker itself and Kubernetes.

4.1.1 Docker in Docker

There are many situations where it would be useful to run Docker inside a Docker container. There are two options to use Docker in Docker. In the *sibling* mode, the Docker running inside the container is the same as the one running on the host. Instead, in the *nested* mode, the container runs a real independent Docker, which starts and manages *child* containers, as opposed to *sibling* containers in the previous case.

The first mode relies on exposing the Docker socket to the container that needs access to it through a volume. Using the Docker CLI, this can be achieved with:

[1] `https://docs.docker.com/docker-hub/official_images/`

```
docker run -v /var/run/docker.sock:/var/run/docker.sock ...
```

However, this mode is not appropriate for OnPIT, since each of the students should have their own instances of Docker and they should be sandboxed.

The second mode does not have such a straightforward implementation since, by default, Docker containers are not able to run a Docker daemon inside. This is because of the restrictions that were put in place to achieve the isolation properties that Docker offers. One such design choice was dropping kernel capabilities that were not needed for general use cases or that were considered dangerous (e.g., `cap_sys_admin` capability, which allows mounting filesystems). Moreover, they are not allowed to access any *cgroup devices*[2]. To overcome these limitations, a container can be started with the *privileged* flag. When this flag is used, the container is started with all the kernel capabilities and without the limitations of the *device* cgroup controller, thereby giving the container almost the same access to the host as if there was no container isolation [4].

Using this flag, a new Docker environment for the students can be created based on an operating system image from Docker Hub. A well-known operating system such as Ubuntu is run with:

```
docker run --privileged -it ubuntu bash
```

This command pulls an image named ubuntu from the Docker Hub registry. Afterwards, it creates and starts a privileged container process, and it creates an interactive bash shell connected to the container.

After getting access to the container, Docker engine (*docker-ce*), the command line interface for the engine (*docker-ce-cli*) and *containerd* are installed inside it.

When the installation is finished, the Docker Daemon is started using the `dockerd` command. To verify the status of the installation the `docker info` command can be used.

In addition, depending on the settings of the Docker running on the host, there might be issues with the container's filesystem. There is an official Docker[3] image on Docker Hub that handles these issues. The specific image tag that does this is dind (standing for Docker in Docker). This image is based on Alpine[4] and in addition to installing the necessary packages for running Docker, it also shares as a volume the directory where Docker stores the containers (i.e., */var/lib/docker*). This container also runs a script that

[2] `https://www.kernel.org/doc/Documentation/cgroup-v1/devices.txt`

[3] `https://hub.docker.com/_/docker`

[4] `https://alpinelinux.org/`

makes sure that the cgroupfs (i.e., kernel's cgroup interface through a filesystem) is properly mounted.

Once the built setup is working, the current state of the container can be saved using the `docker commit` command. This creates a new image that can be saved and pushed to a registry. Alternatively, a Dockerfile that has all the instructions described above can be created and used to build an image.

Using a privileged environment like this is also possible in Kubernetes, therefore this container can be integrated with OnPIT. However, to enable the web server to communicate with the container, the container needs to establish an SSH server to which the web server can connect. Moreover, since this container needs to use the privileged flag and potentially create volumes, this information should be given to the web server to properly configure the container through Kubernetes API Server.

4.1.2 Kubernetes in Docker

There are different ways of setting up a Kubernetes cluster, based on the ease of operation and management of the cluster and available resources. Kubernetes can run on a local machine, on-premises or in the cloud. Typically, a deployment on a local machine is used for creating a learning or testing environment. For such use cases, the Kubernetes community has developed specific tools to make it easier to set up a local cluster that is adapted to the resource constraints of personal machines. Such supported tools include Minikube, kind and K3s. When it comes to production environments, there are many custom solutions based on the cloud provider or the bare metal environment used.

Minikube[5] is the most popular tool for setting up a local Kubernetes cluster. Typically, it runs a single-node Kubernetes cluster inside a virtual machine on the device. The VM driver can be changed using the flag *driver* when starting Minikube. It supports many drivers, such as docker, virtualbox, kvm2, hyperv, vmware and none. The none driver works only on Linux systems with Docker installed and sets up the cluster on the host itself instead of a VM. This driver allows Minikube to run within a Linux container that supports Docker. The Docker in Docker image used previously can serve as a base image for such a Minikube in Docker environment.

kind[6] (Kubernetes in Docker) is a tool that uses Docker containers as nodes for creating a Kubernetes cluster. Its initial purpose was to be used for testing Kubernetes itself. Similarly, kind depends on Docker for creating

[5] `https://minikube.sigs.k8s.io/docs/`

[6] `https://kind.sigs.k8s.io/`

its Kubernetes cluster. Again, running kind inside a container is possible and the same Docker in Docker image built previously can be used as a base image for the new container.

K3s[7] is also a certified Kubernetes distribution, which is designed for running on resource-constrained machines. K3s installation relies on systemd[8] for its features. However, systemd is not part of Docker containers. To use systemd inside a container, the container needs additional capabilities from the operating system of the host. Like with Docker in Docker, the capabilities needed to run systemd and K3s within a container can be enabled by using privileged containers.

In all the cases, kubectl can be installed and used for interaction with the API server through the command line interface.

4.2 Virtual Machine-based Environments

Nowadays, there is a clear distinction between traditional hardware virtualization and the recent OS-level virtualization through containers. Each of these technologies has its own advantages and drawbacks, and users are usually faced with difficult trade-offs when choosing between the two. VMs offer more security and robustness, but that comes at a higher resource utilization. On the other hand, containers offer lower boot time and overhead as well as easier management, but there is the risk of an attacker escaping the sandbox and compromising the host system. This section considers the possibility of creating the student environments with some technologies that try to bring together features of containers and VMs, to narrow the gap between the two.

4.2.1 Kata Containers

Recently, there have been many attempts at building tools that have the advantages of both technologies, with primary focus on the speed of the containers and the security of the VMs [20]. One such attempt is the combination of VMs and containers to create a layered security model by running the containers, which already offer some level of isolation, inside their own VMs as an additional layer. One implementation of this model is Kata Containers[9], an open source container runtime that wraps containers with lightweight virtual machines, thus providing better isolation through hardware virtualization [7]. The container runtime, known as *kata-runtime*, is

[7] `https://k3s.io/`

[8] `https://wiki.archlinux.org/index.php/systemd`

[9] `https://katacontainers.io/`

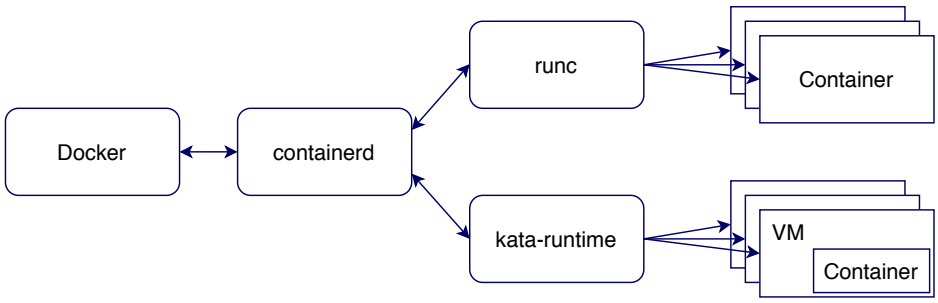


Figure 4.2: Docker components stack with kata-runtime

OCI-compliant. As a result, it works seamlessly with any container engine that supports the OCI runtime specification (e.g., Docker) as well as Kubernetes, since it supports Kubernetes CRI through CRI-O and the containerd CRI plugin. *kata-runtime* creates a virtual machine within which it launches containers or pods, created by either the container engine or Kubernetes kubelet. This enables Kubernetes to transparently choose between the default runtime (such as Docker) and kata-runtime. Figure 4.2 illustrates the Docker stack with runc and kata-runtime running side by side.

Figure 4.3 shows the architecture of kata-runtime and its integration to Docker and Kubernetes (through CRI). Each of the components are described below:

- *kata-runtime* is an OCI-compatible runtime. Thus, it handles all the commands specified by the OCI specification. Additionally, it launches the hypervisor for creating a VM for each container or pod, and the kata-shim instances (described next).

- *kata-shim* is a process that acts as a layer between a container shim, such as containerd-shim or conmon[10], and kata-agent (described next). This is needed since a traditional shim running on the host cannot monitor processes within a VM. Thus, kata-shim acts as the process that the shim can monitor, and it handles all the streams and signals from the shim to the container process.

- *kata-agent* is a process that manages the containers and the processes within them. It relies on libcontainer[11], a library that is also part of runc, to manage the container lifecycle. It communicates with the

[10] `https://github.com/containers/conmon`
[11] `https://github.com/opencontainers/runc/tree/master/libcontainer`

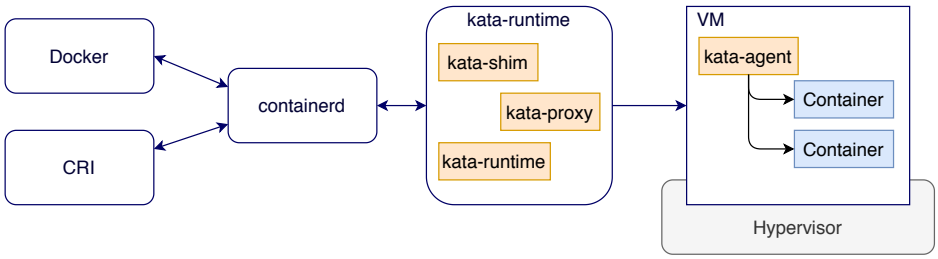


Figure 4.3: Architecture of Kata-containers

other components over gRPC and it can run many containers in the same VM.

The agent process is started by kata-runtime inside the virtual machine, and it runs as a daemon. This agent and kata-runtime communicate through the gRPC protocol. The protocol is used by the runtime to send the container management commands to the agent and to transfer all the input/output streams (stdout, stderr, stdin) between the containers inside the VM and the container engine (e.g., Docker).

The hypervisor boots an operating system image by using a given Linux kernel. Both the kernel and the OS image are optimized for fast boot time and minimal resource utilization, providing only the necessary services to run a container workload. The image is based on Linux and runs only the init daemon (systemd) and the kata-agent. However, kata-runtime allows users to specify a different image, kernel or hypervisor.

- *kata-proxy* is a process that enables kata-agent to access multiple clients on a VM. This is necessary since a VM can run multiple container processes and depending on the way that the guest and host VMs communicate, there might be a need for multiplexing and demultiplexing the I/O streams to and from the containers. The proxy routes the I/O streams and signals between shim instances and the agent. Connection with the agent is done through a Unix domain socket, while the gRPC requests are multiplexed through the yamux[12] library.

Typically, containers have network isolation through namespaces. For instance, by default Docker adds a container to a network that is isolated from the host but shared between the containers. This can however be specified by

[12] `https://github.com/hashicorp/yamux`

users, who have control over the networks available and to which of them a container is attached. Generally, containers implement local Ethernet tunnels through virtual ethernet (veth) devices[13]. These devices are created in pairs: one end is connected to the container's networking namespace, while the other one is connected to the host OS. However, some hypervisors cannot handle veth interfaces; a more common mechanism used by VMs is called TAP[14]. Kata-runtime automatically handles the connection between veth and TAP interfaces.

To use kata-containers with Kubernetes, each of the worker nodes should have kata-runtime installed since a kubelet inside the node relies on a container runtime for managing the containers. Despite its OCI-compliance, kata-runtime cannot distinguish between the pods and containers. By default, a kubelet communicates with the runtime and requests a container creation for each pod or container needed. Upon such a request, kata-runtime creates one VM for each pod and container. The additional information needed to differentiate between pods and containers is provided by the Kubernetes CRI runtime.

One of the main goals of OnPIT is to run student environments with cloud-native technologies, such as Docker and Kubernetes. However, running these technologies in a container-based architecture exposes the host machine to various security threats. Kata Containers improves security, while at the same time allowing the container environments to run as described in the previous section since it seamlessly integrates with Kubernetes. The only modification needed to use Kata Containers is changing the runtime when launching a container.

To minimize the boot time and resource utilization, Kata Containers uses a minimal operating system and kernel for the guest VM. As such, the offered OS and kernel might not fulfill the requirements for running some specific applications and in such situation, custom kernels should be created.

4.2.2 Ignite

Ignite[15] is an open source administration tool for Amazon's Firecracker microVM[16], an open source KVM implementation optimized for high security and isolation through virtual machines, while providing high speed and low resource consumption similar to containers. It was developed for Amazon

[13] `https://man7.org/linux/man-pages/man4/veth.4.html`
[14] `https://www.kernel.org/doc/Documentation/networking/tuntap.txt`
[15] `https://github.com/weaveworks/ignite`
[16] `https://firecracker-microvm.github.io/`

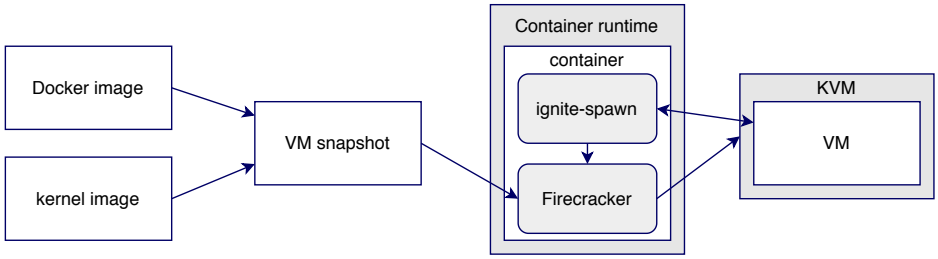


Figure 4.4: Architecture of Ignite

Web Services and is used in the Lambda[17] serverless computing platform and the Fargate[18] serverless compute engine for containers.

Ignite intends to make using Firecracker less challenging for users of container technologies. It achieves this by providing a user experience that resembles that of containers. Specifically, Ignite adapts the docker CLI commands to manage Firecracker VMs the same way Docker manages runc containers. This way, Ignite enables developers to deploy and manage VMs in the same manner as they would do with container workloads. Furthermore, it also automatically handles networking, giving to the VM the same IP that it would take if it were run as a container.

Ignite maintains the start up and shut down speed of Firecracker [16]. To do so, running an image with Ignite boots a new VM through Firecracker. The kernel starts the system initialization (i.e., executes /sbin/init) in the VM and afterwards, Ignite connects the VM to the container networking (through CNI). The image used by Ignite is a Docker container image. However, it does not run as a container within the VM, but instead it runs as a real VM with a dedicated kernel. This abstracts even more the idea of running a container as a VM, since the developer can run the same Docker image as a VM without knowing how to use a specific VMM or how to set up specific settings (e.g., the networking interfaces).

Ignite uses CNI[19] as the default plugin to manage networking [10]. This plugin allows VMs to connect on the same network as the containers when using Docker or Kubernetes, and to map the ports from the VM to the host. However, it does not support communication between multiple nodes. A CNI plugin that supports packet routing between many hosts is needed for that purpose.

Figure 4.4 illustrates the architecture of Ignite. First, the Docker image

[17] `https://aws.amazon.com/lambda/`
[18] `https://aws.amazon.com/fargate/`
[19] `https://github.com/containernetworking/cni`

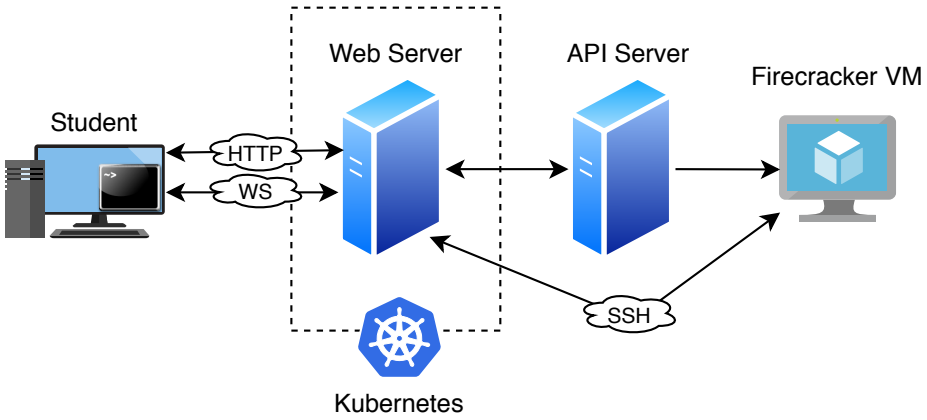


Figure 4.5: Integration of Ignite to OnPIT

is pulled using the container runtime. This image is used to run a container. The root filesystem of this container is exported and together with the image of the kernel they are used to create the snapshot that is going to be used for the VM. Inside the running container, Ignite launches a component called ignite-spawn. This process starts Firecracker, which in turn creates the virtual machine by communicating with KVM. Furthermore, ignite-spawn removes the IP address from the container and gives the same IP to the VM. Finally, ignite-spawn also creates a bridge from the containers veth interface, which is created by the runtime, to the TAP device which is used by Firecracker. This way, all the communication is passed from the container to the VM.

Figure 4.5 shows how Ignite is integrated to OnPIT. To integrate Ignite with OnPIT, the web server needs to initiate the start up and shut down of the VM, and to establish an SSH connection to it. Since Ignite uses virtualization, it cannot exist inside a container, but it has to run on the host (i.e., one of the Kubernetes nodes) or in a different machine. After that, it needs to somehow expose an interface for the web server to send the instructions for managing the lifecycle of the VM (equivalent to the requests for the Kubernetes API). To implement this, the machine that runs Ignite exposes an API server which accepts the requests from the web server and has direct control over Ignite. Upon the creation of a VM, this API server retrieves the IP address and sends it to the web server, which now can directly establish the connection to the VM and forward the input/output streams to the student's browser.

Since the default networking plugin uses the same network as the con-

tainers, by default, the web server is able to communicate with the VMs the same way it does with the container-based solution.

Finally, since Ignite can use any OCI-compliant image, the system allows reusing the same images built for the previous technologies above, and running them directly as VMs instead of containers, without needing further capabilities from the host operating system.

Chapter 5

Evaluation

This chapter evaluates and compares the previously presented technologies in terms of security and interoperability with both OnPIT and the technologies used for the learning environments.

5.1 Security

5.1.1 Containers

Containers share the host's kernel and their isolation is achieved through software virtualization; thus, there is the risk of exposing the host through exploits from within a container. Such exploits lead to breaking the isolation properties of containers and escalating privileges. An example of such an attack is the recent runc vulnerability[1] that allows an attacker to obtain root access on the host. Moreover, if there is a vulnerability on the host kernel, it can be exploited from inside the container, resulting in a direct attack to the host. The best protection against vulnerabilities such as these is to update the host software including the Docker Engine, as well as the Docker images, since they are often patched with protective measures against newly found vulnerabilities.

Another measure is preventing privilege escalation through binaries that run with elevated privileges (i.e., they have setuid or setgid permissions). In Docker, this is done using `--security-opt=no-new-privileges`, while in Kubernetes, it is achieved by setting `allowPrivilegeEscalation` field to `false` in the Security Context.

As mentioned in the Chapter 4, Docker disables by default some capabilities that are considered more dangerous. The most secure model is

[1] `https://nvd.nist.gov/vuln/detail/CVE-2019-5736`

achieved by disabling all of them (with `--cap-drop all`) and then adding (with `--cap-add`) only those needed for each specific container.

When it comes to networking, containers can communicate with each other by default. This feature is disabled by providing the `--icc=false` flag when running the Docker daemon. If some container communication is allowed, it is individually set for each container. In Kubernetes, this is achieved by using Network Policies.

Another important requirement is the availability of the system. To avoid resource exhaustion by a malicious user, Docker allows setting limits on resource usage, such as memory, CPU and maximum number of restarts, file descriptors and processes.

5.1.2 Container Privileges

In Chapter 4 we considered the option of running Docker and Kubernetes inside a Docker container. The first method of using Docker from within a container exposes the Docker socket from the host. Since the owner of this socket is root, giving users of OnPIT access to the socket is the same as giving root access to the host.

The second method needs a privileged container. Privileged containers are spawned because the workloads running within them need system-level permissions. For a use cases such as OnPIT, users already have access to the container through a terminal. These users are considered as potential adversaries: if they are given or somehow gain root access inside the container, they are able to execute code directly on the host with all of the available capabilities, including the previously mentioned `cap_sys_admin`. Since the users already have access to the container through the web terminal, they can find and exploit vulnerabilities within the container to gain root privileges and then try to exploit the host itself. To protect against these privilege escalation attacks, Docker allows using an unprivileged user inside the container, who has lower permissions. The user can be specified at either run time (through the `-u` flag) or build time (by adding the `USER` directive in the Dockerfile). In Kubernetes, the same configuration is achieved by setting the `runAsNonRoot` field to `true` under Security Context.

Even though launching a container with a non-root user is recommended, there are applications that need root privileges. Such an example is the Docker daemon. Using Docker implies running the Docker daemon, which requires root privileges inside the container. At the same time, this process also needs additional capabilities from the host and the container has to be launched as privileged. To allow access from non-root users, Docker creates a group `docker` during installation. The members of this group can build and

execute containers without `sudo`. Despite giving the impression of improved security, this mechanism de-facto gives root access to anyone who belongs to the group. This is because Docker itself still runs as root. For instance, Docker allows sharing any directory of the host with the container, and it does not limit the permissions of the container on such a directory, thereby giving the container full access to it. The command below is an example of how to display the content of the shadow file – which stores the passwords of system users and is only accessible by privileged users – as a non-privileged user by exploiting the privileges of Docker itself:

```
docker run -v /etc:/data ubuntu cat /data/shadow
```

The command creates a shared volume between the host and the newly created container. The volume on the host points at the `etc` directory and is mapped to the `data` directory in the guest container. This guest container can read the data inside `etc` as if it was the owner of the directory, bypassing the permission checks. This could be used by a user of OnPIT, to obtain root access within the container and to tamper with the system,

The Docker community is working on a mode, called rootless, that allows the Docker daemon and containers to run as a non-root user to mitigate potential vulnerabilities in the daemon and the container runtime [14]. However, at this stage the mode is only experimental and has many limitations.

Another option for preventing privilege escalation from within a container even when a root user is used within the container is to re-map this user to a non-root user on the host [6]. The range of UIDs assigned to the user work as normal UIDs within the namespace, but the process has no privileges on the host system. This mode is activated on the daemon with the `--userns-remap` flag. The mode is similar with rootless, but in this case the daemon on the host still runs with root privileges, while in rootless mode both the container and daemon run as non-root users. However, `userns-remap` does not work for system-level workloads because it is not compatible with the `--privileged` flag.

In some scenarios, an application needs additional capabilities (such as Docker) in a container and it also needs to be exposed to untrusted parties. In this case, it is recommended to restrict the capabilities and devices to the minimum needed for the specific container instead of running the container with the `--privileged` flag. For this purpose, Docker provides the `--device`, `--cap-add` and `--cap-drop` flags [4]. These commands give control over individual devices and capabilities, as briefly mentioned above.

Except for Docker, similar challenges and security issues are faced whenever the applications running within a Docker container need additional ca-

pabilities. One such example mentioned in Chapter 4 is `systemd`, which is needed for installing K3s.

5.1.3 VMs

Both VM-based technologies considered in Chapter 4 (i.e., Kata Containers and Ignite) offer improved security over containers.

Kata Containers is built with the specific purpose of providing a security sandbox for containers. All the security considerations mentioned above can also be applied to kata-runtime. However, from a security perspective, kata-runtime is preferable to runc, since it provides another layer of isolation through hardware virtualization and it has its own dedicated kernel. This avoids exposing the host system in case the container is compromised and an attacker manages to escape the isolation properties within it. If a privilege escalation happens, the attacker only gets access to the VM that wraps the container, which provides another layer of defense and reduces the attack surface.

Sharing the resources of the host, such as adding capabilities and accessing host devices, is also supported by kata-runtime. However, this does not give access to the real host machine, but to the guest VM. This allows kata-runtime to be used even with more risky privileges while still being isolated from the host machine.

On the other hand, Ignite relies on Firecracker for handling the isolation. Similarly, Firecracker uses hardware virtualization and its security properties are comparable to other VM technologies. However, since it uses minimal kernel and limited number of emulated devices, its attack surface is also smaller. Furthermore, it is already proven to be secure enough to run multiple tenants on the same hardware in AWS.

5.2 Interoperability

5.2.1 Containers

The container-based environments are the easiest and most straightforward solution when it comes to interoperability with the implemented system. Using containers as student environments takes advantage of the container features and it is fully integrated with the OnPIT system through Kubernetes. This makes it easy to manage and scale based on the actual demand.

OnPIT allows easily incorporation of new courses given a container image and a corresponding Kubernetes object file, if the container has different

requirements. The teachers have a wide range of tools and technologies that come as containerized applications. Many of them have public official images in Docker Hub, which can be extended to create custom environments for the students. Any application that can run as a Docker container, can be integrated to the system. The only needed modification is extending the image by running an SSH server on it and adding the necessary certificate that allows the web server to authenticate against it.

In addition, as previously discussed, containers share most of the operating system features with the host system and it is not possible to run system-level workloads inside a container by default. A workaround to this restriction is running the container with the `--privileged` flag. Indeed, this breaks all the isolation assumptions of Docker, since when a Docker container is launched with this flag, it gets access to all capabilities and all host devices. If such containers are needed, then the specific capabilities and other security settings need to be provided through the Kubernetes' Security and Network Policies. This is an added complexity for the teacher or system administrator and assumes that a teacher can take such decisions that affect the security and availability of the whole system, which is not desirable.

5.2.2 VMs

Using VM-based solutions with OnPIT require installation of the technologies alongside the existing Kubernetes cluster. This brings up the need to set up a custom cluster of machines that is used for the Kubernetes cluster and the additional technologies, instead of utilizing a Kubernetes implementation from the cloud provider (e.g., the Google Kubernetes Engine).

Both Kata Containers and Ignite are straightforward to install and set up.

The runtime of Kata Containers is OCI-compliant, and this enables it to seamlessly plug in to any container engine that supports the standard. It also supports the Kubernetes CRI, allowing integration to Kubernetes for orchestration. This makes its integration to OnPIT seamless, since after installing the runtime in each worker node, the only step needed to switch between runc and kata-runtime is setting the `io.kubernetes.cri.untrusted-workload` annotation for the specific container to `true`. kata-runtime also supports sharing the resources of the host, such as adding capabilities and accessing host devices. However, this does not give access to the real host machine, but to the guest VM. This allows OnPIT to rely on the hypervisor for providing isolation between the containers, even for those with more dangerous capabilities. On the other hand, Ignite does not plug into Docker or Kubernetes and needs to be used as an independent tool – at least, with the current im-

plementation. However, compared to traditional hypervisors, Ignite provides a familiar CLI that resembles the Docker CLI, and it utilizes OCI-compliant images to create the VM it starts. Ignite also transparently attaches to the same network interface as containers, allowing communication between the containers and the VMs.

The difference between Kata Containers and Ignite is that Ignite takes a Docker image and runs it as a real VM. Using Ignite for the learning environments means that the user has direct access to a virtual machine, and it does not run containers inside the VM (unless a runtime is installed to be used by the user). Since Ignite runs a Docker image as a VM, all the container settings needed to run system-level workloads inside containers are no longer necessary, since the student environment is a traditional virtual machine and not a container.

Both these solutions make use of light virtual machines to improve the boot time and resource overhead. Ignite achieves this by utilizing Firecracker, while Kata Containers allows selecting different hypervisors, including (but not limited to) Firecracker. However, an instance of Firecracker with Kata Containers only serves as a wrapper, since the OCI image is spawned as a container inside it.

Firecracker is capable of running thousands of VMs on the same hardware. Its memory overhead is as small as 3% and allows high density of VMs, while the CPU overhead is even smaller [18]. Moreover, its boot time is as little as 150ms. Even though it runs a minimal Linux system, it runs any software that does not have specific hardware requirements and it also allows hosts to configure memory and CPU cores exposed to a VM.

On the other hand, the difference on boot time between kata-runtime and runc is negligible if the VM is started with only one virtual CPU [35]. This is the default setting, and increasing it impacts the VM's boot time and the memory footprint.

Due to architectural differences between container runtimes and Kata Containers, the latter has some limitations that should be taken into consideration [8]:

- it does not support Docker host network. As a result, it is not possible to directly access the host network from within the VM;
- it does not support network namespace sharing between containers;
- it does not support the Security Enhanced Linux (SELinux);
- it is not straightforward to apply resource limitations. Sometimes the constraints need to be applied at both container and VM level.

Chapter 6

Conclusion

Learning computer science is facilitated with hands-on exercises through interactive tutorials on different topics. This thesis presented the design and implementation of OnPIT, an online platform for interactive tutorials. The platform is accessible online through a web browser and gives access to a dedicated virtual learning environment. The learning environment provides pre-installed software tools needed for a tutorial and is isolated from the other instances. OnPIT allows teachers to create their own content and extend the basic functionality of labs with new customized environments. Additionally, the platform implements authentication/authorization mechanisms and enables teachers to specify rules for automated progress tracking.

The platform is built with a container architecture based on Docker and Kubernetes. This thesis evaluated the feasibility of using Docker containers as isolated learning environments for the students. When it comes to the supported content, the container-based implementation allows extension of environments by utilizing any container image supported by Docker and Kubernetes. However, there are applications that by default cannot run inside containers. The thesis presented the limitations of Docker containers by evaluating the feasibility of running Docker and Kubernetes inside a Docker container. This was achieved by obtaining additional privileges and capabilities from the host operating system.

However, running containers with additional capabilities opens more security issues in terms of isolation, especially if they are not configured properly or give access to untrusted users. To overcome these issues, this work presented alternative solutions that combine containers with VMs to provide the advantages of both. First, it presented Kata Containers, a container runtime that runs containers inside their own VMs. Next, it presented Ignite and Firecracker, which allow Docker images to run as VMs with minimal kernels and minimal operating systems instead of containers. The thesis presented

their architecture and showed how they are integrated into the platform. Furthermore, it showed that these technologies can be used instead of Docker containers since the performance overhead is negligible when using their default options. These solutions also provide improved isolation and security through hardware virtualization. Kata Containers allows running containers with additional privileges and capabilities, while giving access to these features from the guest VM, thus maintaining the isolation properties. Similarly, Firecracker relies on virtualization technology for its isolation, and since it has its own kernel and operating system, the applications running within the VM can get direct access to all the features of the operating system.

Future work could address the direct integration of Firecracker VMs into Kubernetes. Furthermore, the resource limitation of VMs could be dynamically changed based on the needs of every instance. In addition, OnPIT could be extended to allow students to run multiple environments at the same time and integrate other features on the browser, such as a text editor, to make the work easier.

Bibliography

[1] A Practical Introduction to Container Terminology. https://developers.redhat.com/blog/2018/02/22/container-terminology-practical-introduction/. Accessed: 2020-07-15.

[2] About storage drivers. https://docs.docker.com/storage/storagedriver/. Accessed: 2020-07-15.

[3] Docker Overview. https://docs.docker.com/get-started/overview/. Accessed: 2020-07-01.

[4] Docker run reference - Runtime privilege and Linux capabilities. https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities. Accessed: 2020-07-15.

[5] Docker security. https://docs.docker.com/engine/security/security/. Accessed: 2020-07-01.

[6] Isolate containers with a user namespace. https://docs.docker.com/engine/security/userns-remap/. Accessed: 2020-07-15.

[7] Kata Containers Architecture. https://github.com/kata-containers/documentation/blob/master/design/architecture.md. Accessed: 2020-07-15.

[8] Kata Containers Limitation. https://github.com/kata-containers/documentation/blob/master/Limitations.md. Accessed: 2020-07-15.

[9] Kubernetes Components. https://kubernetes.io/docs/concepts/overview/components/. Accessed: 2020-07-01.

[10] Networking - Weave Ignite. https://ignite.readthedocs.io/en/stable/networking/. Accessed: 2020-07-15.

[11] ONPIT: Online Platform for Interactive Tutorials. `https://onlinelearning.aalto.fi/pilot/onpit-online-platform-for-interactive-tutorials`. Accessed: 2020-07-15.

[12] Overview of Cloud Native Security. `https://kubernetes.io/docs/concepts/security/overview/`. Accessed: 2020-07-01.

[13] Pods. `https://kubernetes.io/docs/concepts/workloads/pods/`. Accessed: 2020-07-15.

[14] Run the Docker daemon as a non-root user (Rootless mode). `https://docs.docker.com/engine/security/rootless/`. Accessed: 2020-07-15.

[15] The Kubernetes API. `https://kubernetes.io/docs/concepts/overview/kubernetes-api/`. Accessed: 2020-07-01.

[16] Weave Ignite. `https://github.com/weaveworks/ignite`. Accessed: 2020-07-15.

[17] What is Kubernetes. `https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/`. Accessed: 2020-07-01.

[18] Agache, A., Brooker, M., Iordache, A., Liguori, A., Neugebauer, R., Piwonka, P., and Popa, D.-M. Firecracker: Lightweight virtualization for serverless applications. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)* (2020), pp. 419–434.

[19] Ally, M. Foundations of educational theory for online learning. *Theory and practice of online learning 2* (2004), 15–44.

[20] Arnautov, S., Trach, B., Gregor, F., Knauth, T., Martin, A., Priebe, C., Lind, J., Muthukumaran, D., O'keeffe, D., Stillwell, M. L., et al. {SCONE}: Secure linux containers with intel {SGX}. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)* (2016), pp. 689–703.

[21] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., and Warfield, A. Xen and the art of virtualization. *ACM SIGOPS operating systems review 37*, 5 (2003), 164–177.

[22] Bellard, F. Qemu, a fast and portable dynamic translator. In *USENIX Annual Technical Conference, FREENIX Track* (2005), vol. 41, p. 46.

[23] Bernstein, D. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing 1*, 3 (2014), 81–84.

[24] Brendan Burns, C. T. *Managing Kubernetes*. O'Reilly Media, Inc., 2018.

[25] Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. Borg, omega, and kubernetes. *Queue 14*, 1 (2016), 70–93.

[26] Fernández, L., Andersson, R., Hagenrud, H., Korhonen, T., Laface, E., Zupanc, B., et al. Jupyterhub at the ess. an interactive python computing environment for scientists and engineers. In *This conference* (2016).

[27] Galán, F., Fernández, D., Ruiz, J., Walid, O., and de Miguel, T. Use of virtualization tools in computer network laboratories. In *Information Technology Based Proceedings of the FIfth International Conference onHigher Education and Training, 2004. ITHET 2004.* (2004), IEEE, pp. 209–214.

[28] Gaspar, A., Langevin, S., Armitage, W., Sekar, R., and Daniels, T. The role of virtualization in computing education. *ACM sIGCsE bulletin 40*, 1 (2008), 131–132.

[29] Honeycutt, J. Microsoft virtual pc 2004 technical overview. *Microsoft, Nov* (2003).

[30] Lunsford, D. L. Virtualization technologies in information systems education. *Journal of Information Systems Education 20*, 3 (2009), 339.

[31] Marston, S., Li, Z., Bandyopadhyay, S., Zhang, J., and Ghalsasi, A. Cloud computingâthe business perspective. *Decision support systems 51*, 1 (2011), 176–189.

[32] Muller, A., and Wilson, S. Virtualization with vmware esx server.

[33] Pearce, M., Zeadally, S., and Hunt, R. Virtualization: Issues, security threats, and solutions. *ACM Computing Surveys (CSUR) 45*, 2 (2013), 1–39.

[34] Popek, G. J., and Goldberg, R. P. Formal requirements for virtualizable third generation architectures. *Communications of the ACM 17*, 7 (1974), 412–421.

[35] Randazzo, A., and Tinnirello, I. Kata containers: An emerging architecture for enabling mec services in fast and secure way. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)* (2019), IEEE, pp. 209–214.

[36] Robin, J. S., and Irvine, C. E. Analysis of the intel pentium's ability to support a secure virtual machine monitor. Tech. rep., NAVAL POSTGRADUATE SCHOOL MONTEREY CA DEPT OF COMPUTER SCIENCE, 2000.

[37] Sugerman, J., Venkitachalam, G., and Lim, B.-H. Virtualizing i/o devices on vmware workstation's hosted virtual machine monitor. In *USENIX Annual Technical Conference, General Track* (2001), pp. 1–14.

[38] Velte, A., and Velte, T. *Microsoft virtualization with Hyper-V*. McGraw-Hill, Inc., 2009.

[39] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and De Rose, C. A. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing* (2013), IEEE, pp. 233–240.

[40] Zhao, J., and Forouraghi, B. An interactive and personalized cloud-based virtual learning system to teach computer science. In *International Conference on Web-Based Learning* (2013), Springer, pp. 101–110.

Appendix A

Database Schema

This appendix shows the schema of the database which consists of the following tables: Course (stores the content of courses), Labs (stores the content of labs), Grade (stores the progress of students on the labs), User (stores the information of the users), CourseOwnership (stores the roles of users for each course) and CourseToLab (stores the relations between courses and labs).

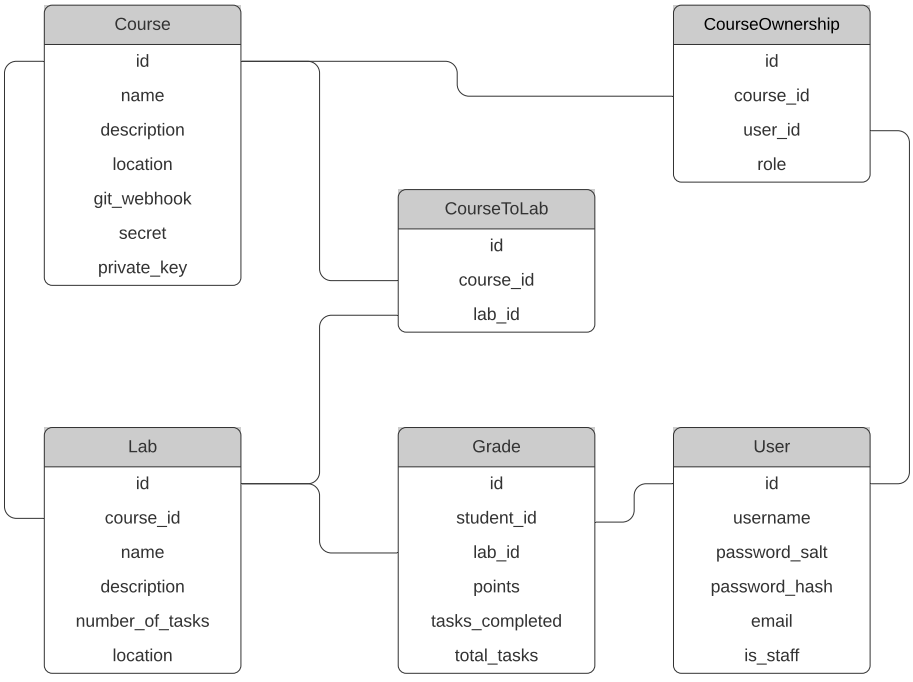


Figure A.1: Schema of the database